Pilares da POO – Encapsulamento

UNIVERSIDADE DO OESTE DE SANTA CATARINA - UNOESC

Ciência da Computação | Programação II

Prof.: Leandro Otavio Cordova Vieira

Objetivos da Aula

1

Entender Encapsulamento

Compreender o conceito fundamental do encapsulamento como um dos quatro pilares da Programação Orientada a Objetos e sua importância no desenvolvimento de software.

2

Aplicar Modificadores de Acesso

Aprender a utilizar corretamente os modificadores de acesso (public, private, protected) para controlar a visibilidade de atributos e métodos em PHP.

3

Implementar Getters e Setters

Desenvolver habilidades para criar e implementar métodos acessores e modificadores para manipular dados de forma controlada.

4

Exercícios Práticos

Aplicar os conceitos aprendidos em atividades práticas para fixação do conteúdo e desenvolvimento de habilidades de programação.

O que é Encapsulamento?

Definição

Encapsulamento é o princípio de esconder os detalhes internos de um objeto e fornecer uma interface controlada para interagir com ele. É como uma "cápsula" que protege os dados de serem acessados ou modificados incorretamente.

Motivação

Surgiu da necessidade de controlar o acesso aos dados, evitando manipulações inconsistentes que podem comprometer a integridade do sistema.

Benefícios

- Maior segurança dos dados
- Redução de dependências entre classes
- Controle de acesso e validações
- Facilidade de manutenção

Problema sem Encapsulamento

Acesso Direto aos Atributos

Sem encapsulamento, os atributos ficam expostos e podem ser modificados livremente por qualquer parte do código.

```
class ContaBancaria {
    public $saldo = 1000;
}

$conta = new ContaBancaria();

// Qualquer código pode alterar o saldo diretamente
$conta->saldo = -5000; // Saldo negativo!

echo "Saldo: R$ " . $conta->saldo; // Saldo: R$ -5000
```

Problemas:

- Alterações sem validação (saldo negativo)
- Impossibilidade de rastrear modificações
- Não há controle sobre as operações

Solução com Encapsulamento

```
class ContaBancaria {
    private $saldo = 1000;

    public function sacar($valor) {
        if ($valor > 0 && $valor <= $this->saldo) {
            $this->saldo -= $valor;
            return true;
        }
        return false;
    }

    public function getSaldo() {
        return $this->saldo;
    }
}

$conta = new ContaBancaria();
$sucesso = $conta->sacar(500);

echo "Saldo: R$ " . $conta->getSaldo(); // Saldo: R$ 500
// $conta->saldo = -5000; // Erro! Propriedade privada
```

Benefícios:

- Validação antes de modificar o saldo
- Impossibilidade de alterar o saldo diretamente
- Comportamento previsível e controlado

Modificadores de Acesso em PHP

public

Acesso de qualquer lugar, tanto dentro quanto fora da classe.

Uso: APIs públicas, métodos que devem ser acessíveis externamente.

```
public $nome;
```

private

Acesso apenas dentro da própria classe. Nem classes filhas podem acessar.

Uso: Dados sensíveis, implementações internas.

```
private $senha;
```

protected

Acesso dentro da própria classe e classes que herdam dela.

Uso: Atributos que classes filhas precisam acessar.

```
protected $desconto;
```

A escolha do modificador adequado é fundamental para o encapsulamento eficiente e para a segurança dos seus dados.

Exemplo em PHP – public

```php
class Produto {
    public $nome;
    public $preco;

    public function __construct($nome, $preco) {
        $this->nome = $nome;
        $this->preco = $preco;
    }
}

$celular = new Produto("Smartphone", 1500);

// Acesso direto aos atributos públicos
$celular->nome = "Smartphone Premium";
$celular->preco = 2000;

echo $celular->nome; // Smartphone Premium
echo $celular->preco; // 2000
```

Características

- Acesso total de qualquer parte do código
- Modificação direta sem validações
- Facilidade de uso, mas menos seguro
- Quebra o princípio de encapsulamento

Casos de uso adequados:

- Constantes
- Métodos utilitários
- APIs intencionalmente públicas

Exemplo em PHP – private

```php
class ContaBancaria {
    private $saldo;
    private $numeroConta;

    public function __construct($numeroConta, $saldoInicial) {
        $this->numeroConta = $numeroConta;
        $this->saldo = $saldoInicial;
    }

    public function depositar($valor) {
        if ($valor > 0) {
            $this->saldo += $valor;
            return true;
        }
        return false;
    }

    public function getSaldo() {
        return $this->saldo;
    }
}

$conta = new ContaBancaria("12345-6", 1000);
$conta->depositar(500);

// Erro! Atributo privado
// $conta->saldo = -1000;

echo $conta->getSaldo(); // 1500
```

Características

- Acesso restrito apenas à própria classe
- Impede modificações externas diretas
- Oferece maior controle e segurança
- Implementa verdadeiro encapsulamento

Casos de uso adequados:

- Dados sensíveis (senhas, saldos)
- Implementações internas
- Estados que precisam de validação

Exemplo em PHP – protected

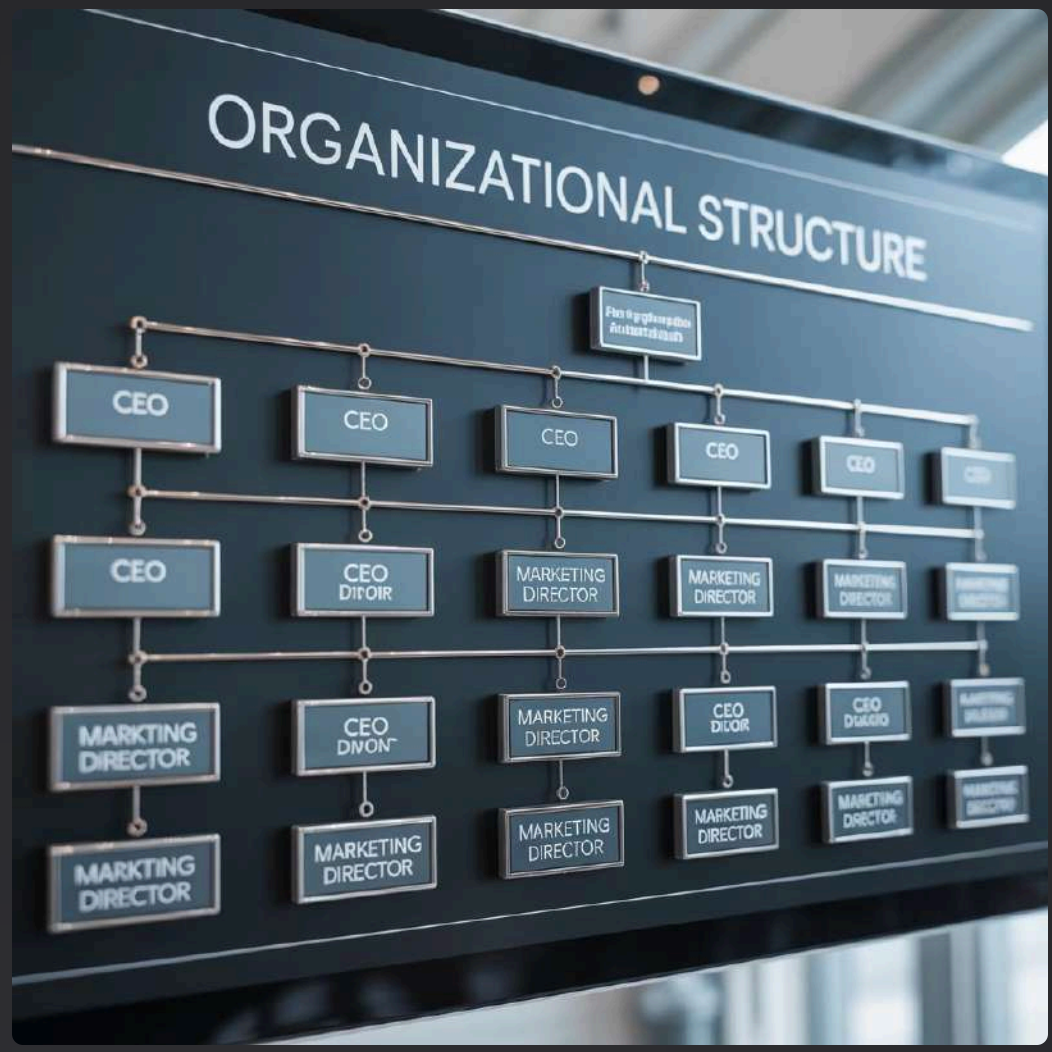
```php
class Funcionario {
    protected $salario;
    protected $nome;

    public function __construct($nome, $salario) {
        $this->nome = $nome;
        $this->salario = $salario;
    }

    public function getSalario() {
        return $this->salario;
    }
}

class Gerente extends Funcionario {
    private $bonus;

    public function __construct($nome, $salario, $bonus) {
        parent::__construct($nome, $salario);
        $this->bonus = $bonus;
    }

    public function getSalarioTotal() {
        // Acesso ao atributo protegido da classe pai
        return $this->salario + $this->bonus;
    }
}

$gerente = new Gerente("Ana", 5000, 1000);
echo $gerente->getSalarioTotal(); // 6000
```

Características

- Acesso dentro da classe e suas subclasses
- Facilita a herança e extensão
- Equilíbrio entre proteção e flexibilidade

Casos de uso adequados:

- Atributos que classes filhas precisam acessar
- Métodos que serão sobrescritos
- Implementações que serão estendidas



Getters e Setters

Definição

Métodos especiais para acessar (getters) e modificar (setters) atributos privados ou protegidos de uma classe.

Convenções de Nomenclatura

- `getNomeAtributo()` - para obter o valor
- `setNomeAtributo($valor)` - para definir o valor
- `isNomeAtributo()` - para atributos booleanos

Papel no Encapsulamento

- Permitem acesso controlado aos atributos
- Implementam validações antes de modificar valores
- Permitem transformações de dados
- Mantêm a compatibilidade se a implementação interna mudar

Getters e setters são como os porteiros de um prédio: controlam quem entra e sai, e sob quais condições, garantindo a segurança e integridade dos dados.

Exemplo – Getters

```
class Produto {
    private $preco;
    private $estoque;
    private $nome;

    public function __construct($nome, $preco, $estoque) {
        $this->nome = $nome;
        $this->preco = $preco;
        $this->estoque = $estoque;
    }

    // Getter simples
    public function getNome() {
        return $this->nome;
    }

    // Getter com formatação
    public function getPreco() {
        return "R$ " . number_format($this->preco, 2, ',', '.');
    }

    // Getter com lógica condicional
    public function getDisponibilidade() {
        return ($this->estoque > 0) ? "Disponível" : "Indisponível";
    }

    // Getter que retorna valor calculado
    public function getValorTotalEstoque() {
        return $this->preco * $this->estoque;
    }
}

$produto = new Produto("Notebook", 3500, 10);
echo $produto->getNome(); // Notebook
echo $produto->getPreco(); // R$ 3.500,00
echo $produto->getDisponibilidade(); // Disponível
echo $produto->getValorTotalEstoque(); // 35000
```

Os getters não apenas retornam valores, mas podem transformá-los, formatá-los ou combiná-los, oferecendo uma interface rica para acesso aos dados.

Exemplo – Setters

```
class Produto {
    private $preco;
    private $estoque;
    private $nome;

    // Setter com validação simples
    public function setNome($nome) {
        if (strlen($nome) >= 3) {
            $this->nome = $nome;
            return true;
        }
        return false;
    }

    // Setter com validação mais complexa
    public function setPreco($preco) {
        if ($preco > 0) {
            $this->preco = $preco;
            return true;
        } else {
            throw new Exception("Preço deve ser maior que zero");
        }
    }

    // Setter com transformação de dados
    public function setEstoque($quantidade) {
        $this->estoque = max(0, intval($quantidade));
    }

    // Setter com lógica de negócio
    public function reduzirEstoque($quantidade) {
        if ($quantidade > 0 && $quantidade <= $this->estoque) {
            $this->estoque -= $quantidade;
            return true;
        }
        return false;
    }
}

$produto = new Produto();
$produto->setNome("Notebook"); // true
$produto->setPreco(3500); // true
$produto->setEstoque(10); // Estoque = 10
$produto->reduzirEstoque(3); // true, Estoque = 7
```

Setters permitem validações e transformações antes de modificar os dados, garantindo a integridade do estado do objeto.

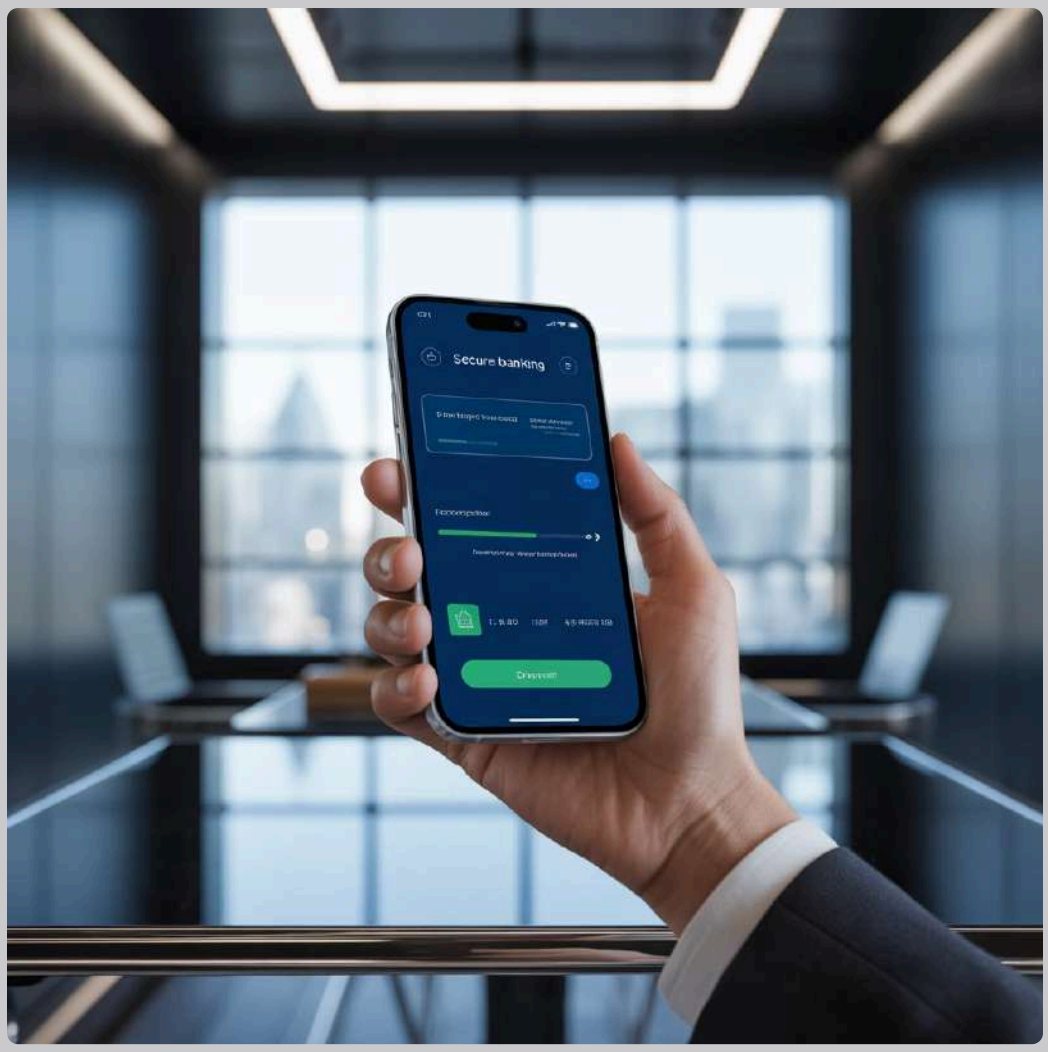
Cenário Real: Banco Digital

Problema

Um banco digital precisa garantir que operações financeiras sejam seguras e que o saldo das contas não possa ser manipulado diretamente.

Solução com Encapsulamento

```
class ContaBancaria {
    private $saldo;
    private $numero;
    private $historico = [];

    public function __construct($numero, $saldoInicial = 0) {
        $this->numero = $numero;
        $this->saldo = $saldoInicial;
        $this->registrarOperacao("Abertura", $saldoInicial);
    }

    public function depositar($valor) {
        if ($valor <= 0) {
            return false;
        }

        $this->saldo += $valor;
        $this->registrarOperacao("Depósito", $valor);
        return true;
    }

    public function sacar($valor) {
        if ($valor <= 0 || $valor > $this->saldo) {
            return false;
        }

        $this->saldo -= $valor;
        $this->registrarOperacao("Saque", $valor);
        return true;
    }

    private function registrarOperacao($tipo, $valor) {
        $this->historico[] = [
            'data' => date('Y-m-d H:i:s'),
            'tipo' => $tipo,
            'valor' => $valor,
            'saldo' => $this->saldo
        ];
    }

    public function getSaldo() {
        return $this->saldo;
    }

    public function getHistorico() {
        return $this->historico;
    }
}
```



Benefícios do Encapsulamento

- Saldo protegido por ser `private`
- Todas as operações são validadas
- Histórico de transações é registrado automaticamente
- Impossível realizar operações inválidas (saques maiores que o saldo)
- Interface clara para operações bancárias

O encapsulamento garante que operações financeiras sigam regras rigorosas, evitando inconsistências nos dados bancários.

Cenário Real: E-commerce

```
class ProdutoLoja {
 private $nome;
 private $preco;
 private $precoPromocional = null;
 private $estoque;
 private $categoria;

 public function __construct($nome, $preco, $estoque, $categoria) {
 $this->nome = $nome;
 $this->setPreco($preco);
 $this->estoque = $estoque;
 $this->categoria = $categoria;
 }

 public function setPreco($valor) {
 if ($valor <= 0) {
 throw new Exception("Preço deve ser maior que zero");
 }
 $this->preco = $valor;

 // Recalcula preço promocional se existir
 if ($this->precoPromocional !== null) {
 $this->aplicarDesconto($this->getPercentualDesconto());
 }
 }

 public function aplicarDesconto($percentual) {
 if ($percentual < 0 || $percentual > 90) {
 throw new Exception("Desconto deve estar entre 0% e 90%");
 }

 $this->precoPromocional = $this->preco * (1 - ($percentual / 100));
 return $this->precoPromocional;
 }

 public function getPercentualDesconto() {
 if ($this->precoPromocional === null) {
 return 0;
 }

 return (($this->preco - $this->precoPromocional) / $this->preco) * 100;
 }

 public function getPreco() {
 return ($this->precoPromocional !== null) ? $this->precoPromocional : $this->preco;
 }

 public function getPrecoOriginal() {
 return $this->preco;
 }
}

// Uso
$produto = new ProdutoLoja("Smartphone", 1500, 10, "Eletrônicos");
echo $produto->getPreco(); // 1500

$produto->aplicarDesconto(20);
echo $produto->getPreco(); // 1200

// Atualiza o preço original, o desconto percentual é mantido
$produto->setPreco(2000);
echo $produto->getPreco(); // 1600 (20% de desconto aplicado automaticamente)
```

Cenário Real: Cadastro de Usuário

```php
class Usuario {
    private $id;
    private $nome;
    private $email;
    private $senha;
    private $tentativasLogin = 0;
    private $bloqueado = false;

    public function __construct($nome, $email, $senha) {
        $this->nome = $nome;
        $this->email = $email;
        $this->setSenha($senha);
    }

    public function setSenha($senha) {
        if (strlen($senha) < 8) {
            throw new Exception("Senha deve ter pelo menos 8 caracteres");
        }

        // Nunca armazena a senha em texto puro
        $this->senha = password_hash($senha, PASSWORD_DEFAULT);
    }

    public function verificarSenha($senhaDigitada) {
        if ($this->bloqueado) {
            return false;
        }

        $senhaCorreta = password_verify($senhaDigitada, $this->senha);

        if (!$senhaCorreta) {
            $this->tentativasLogin++;

            if ($this->tentativasLogin >= 5) {
                $this->bloqueado = true;
            }

            return false;
        }

        // Reset de tentativas após login bem-sucedido
        $this->tentativasLogin = 0;
        return true;
    }

    public function estaBloqueado() {
        return $this->bloqueado;
    }

    public function desbloquear() {
        $this->bloqueado = false;
        $this->tentativasLogin = 0;
    }
}

// Uso
$usuario = new Usuario("Carlos Silva", "carlos@email.com", "Senha@123");

$loginOk = $usuario->verificarSenha("senha123"); // false
// Após 5 tentativas erradas...
echo $usuario->estaBloqueado(); // true
```

O encapsulamento permite:

- Proteger senhas com hash
- Validar requisitos de segurança
- Implementar bloqueio após múltiplas tentativas
- Impedir acesso direto a dados sensíveis

Atividade em Sala

Refatorando a Classe Produto

Antes (sem encapsulamento)

```php
class Produto {
    public $id;
    public $nome;
    public $preco;
    public $quantidade;

    public function __construct($id, $nome, $preco, $quantidade) {
        $this->id = $id;
        $this->nome = $nome;
        $this->preco = $preco;
        $this->quantidade = $quantidade;
    }

    public function calcularTotal() {
        return $this->preco * $this->quantidade;
    }
}
```

Depois (com encapsulamento)

```php
class Produto {
    private $id;
    private $nome;
    private $preco;
    private $quantidade;

    public function __construct($id, $nome, $preco, $quantidade) {
        $this->id = $id;
        $this->setNome($nome);
        $this->setPreco($preco);
        $this->setQuantidade($quantidade);
    }

    public function getId() {
        return $this->id;
    }

    public function getNome() {
        return $this->nome;
    }

    public function setNome($nome) {
        if (empty($nome)) {
            throw new Exception("Nome não pode ser vazio");
        }
        $this->nome = $nome;
    }

    // Adicione os demais getters e setters
}
```

Tarefa

Complete a classe Produto adicionando os getters e setters faltantes para preço e quantidade, incluindo validações apropriadas (preço positivo, quantidade não negativa).

Atividade em Pares

Revisão Cruzada de Código

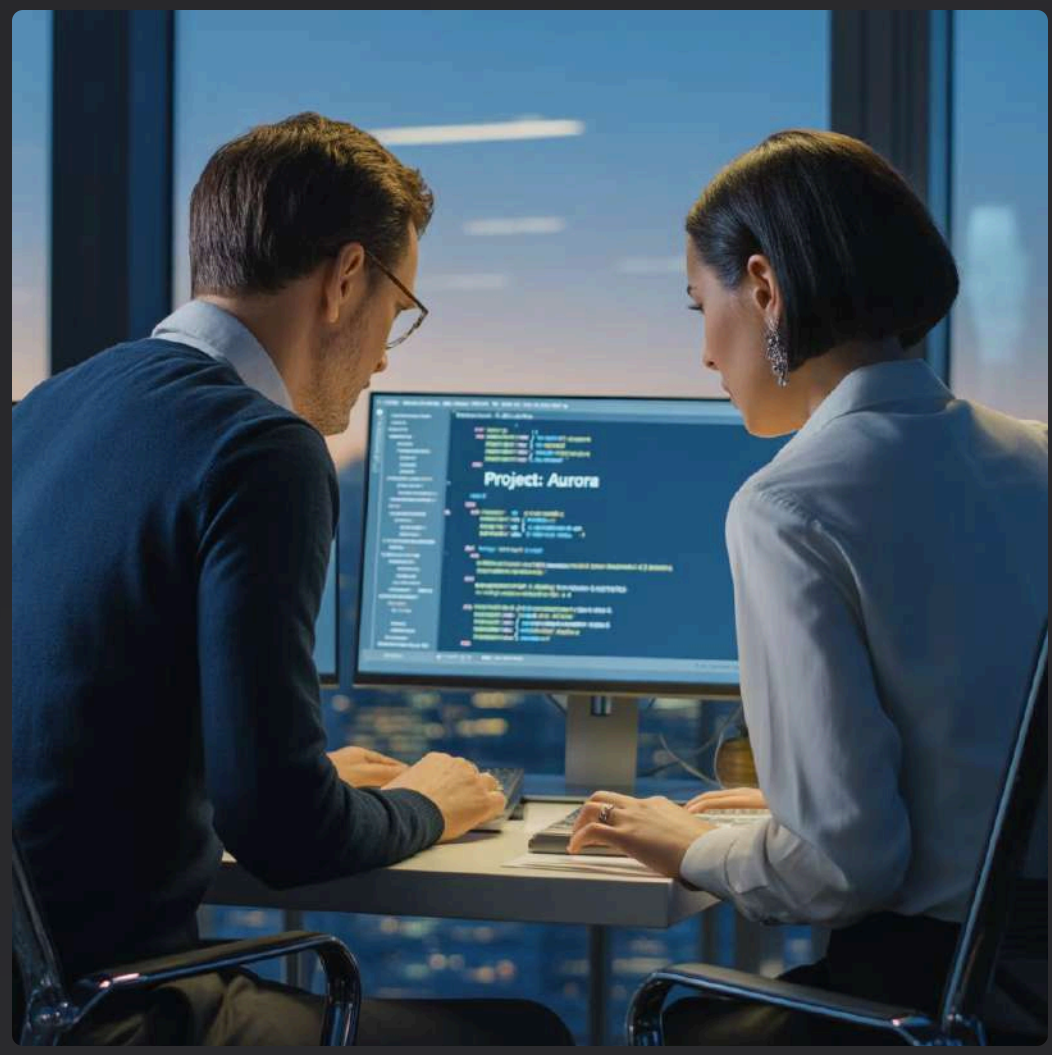
Formem duplas e analisem o código um do outro, verificando:

- Os atributos estão devidamente encapsulados?
- Os getters e setters implementam validações adequadas?
- O código segue boas práticas de encapsulamento?
- Há casos em que métodos específicos seriam melhores que getters/setters genéricos?

Desafio Extra

Implemente uma classe CarrinhoDeCompras que utilize a classe Produto encapsulada para:

- Adicionar/remover produtos
- Calcular total do carrinho
- Aplicar descontos
- Verificar disponibilidade em estoque



Feedback Estruturado

Ao revisar o código do colega, forneça feedback seguindo esta estrutura:

1. Pontos positivos: O que foi bem implementado?
2. Oportunidades de melhoria: O que poderia ser aprimorado?
3. Sugestões: Como implementar essas melhorias?

Use um formulário de revisão para garantir que todos os aspectos importantes foram verificados.

Boas Práticas de Encapsulamento

Princípio do "Menor Privilégio Possível"

Sempre defina atributos com o nível mais restritivo de acesso possível. Na dúvida, comece com `private` e relaxe a restrição apenas se necessário.

Evite Getters/Setters Desnecessários

Não crie getters e setters automaticamente para todos os atributos. Questione se a exposição é realmente necessária. Métodos específicos de negócio são frequentemente melhores que getters/setters genéricos.

Valide no Ponto de Entrada

Sempre valide os dados nos setters antes de modificar os atributos. Esta é a primeira linha de defesa contra dados inválidos.

Métodos Específicos vs. Setters Genéricos

Para operações de negócio, prefira métodos com nomes significativos como `depositar()` e `sacar()` em vez de `setSaldo()`, pois eles comunicam melhor a intenção.

Imutabilidade Quando Possível

Se um atributo não deve mudar após a criação do objeto, não crie um setter para ele. Defina-o apenas no construtor.

Evite "Vazamento" da Implementação

Não exponha detalhes de implementação interna através da sua interface pública. Mantenha a abstração consistente.

Seguir estas práticas resultará em código mais robusto, manutenível e menos propenso a bugs relacionados a estados inconsistentes.

Resumo da Aula

Conceito de Encapsulamento

Princípio de ocultar os detalhes internos de implementação e expor apenas uma interface controlada para interagir com o objeto.

Modificadores de Acesso

- `public`: Acessível de qualquer lugar
- `private`: Acessível apenas dentro da própria classe
- `protected`: Acessível na classe e em suas subclasses

Getters e Setters

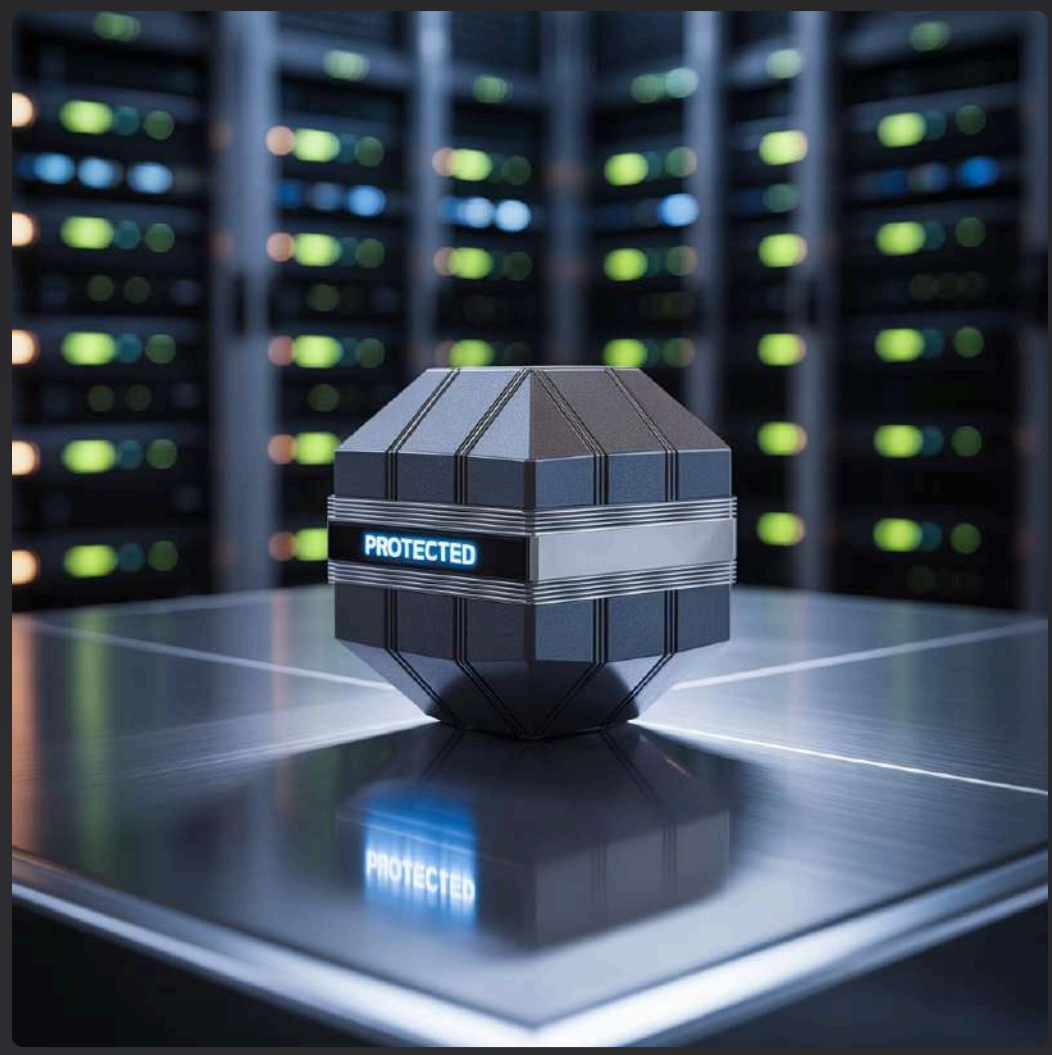
Métodos que controlam o acesso e a modificação de atributos privados, permitindo validações e transformações.

Exemplos Práticos

- Banco Digital: Controle de saldo e operações
- E-commerce: Gestão de produtos e preços
- Cadastro: Segurança de informações sensíveis

Boas Práticas

- Menor privilégio possível
- Validação nos pontos de entrada
- Métodos específicos de negócio
- Imutabilidade quando apropriado



Próxima Aula: Herança em PHP

Conceito de Herança

Aprenderemos como criar hierarquias de classes e reutilizar código através do segundo pilar da POO: a Herança.

Classes Pai e Filha

Implementação de classes base e derivadas, aproveitando comportamentos existentes e estendendo-os.

Desenvolvimento Incremental

Como criar sistemas complexos de forma gradual, reaproveitando código e reduzindo duplicação.

Prepare-se revisando os conceitos de encapsulamento vistos hoje, pois eles serão fundamentais para compreender como a herança funciona em conjunto com o controle de acesso aos atributos e métodos.